# Inventory management system

## Part 01 Features:

* **Assumptions:**
  + **Operating Environment:**

The system is designed for a large-scale warehouse, impacting feature prioritization and system scalability.

* + **User Authentication and Access Control:**

Robust user authentication and privilege management to ensure secure and protected access to the system.

* + **Product Search Functionality:**

Comprehensive product search capabilities, including searches across sold products, ordered products, and items currently in the inventory.

* + **Efficient Product Addition:**

Streamlined methods for adding products to the inventory, including options such as camera-based input, barcode scanning, and manual entry. The system supports multiple product categories.

* + **Purchase Record Management:**

Maintenance of detailed records for products purchased from third-party suppliers, ensuring a transparent overview of procurement activities.

* + **Sales Tracking:**

Active sales management module to track and record all sales transactions, providing real-time insights into product movement.

* + **Order Processing:**

Order processing functionality for both purchases and sales, allowing for the seamless management of orders in various stages of completion.

* + **Record Printing and Documentation:**

Comprehensive record maintenance with detailed information on incoming and outgoing inventory. This feature supports the generation of printable reports for a comprehensive overview.

* + **Administrator Privileges:**

Implementation of an administrative role with ultimate privileges for efficient system management and control.

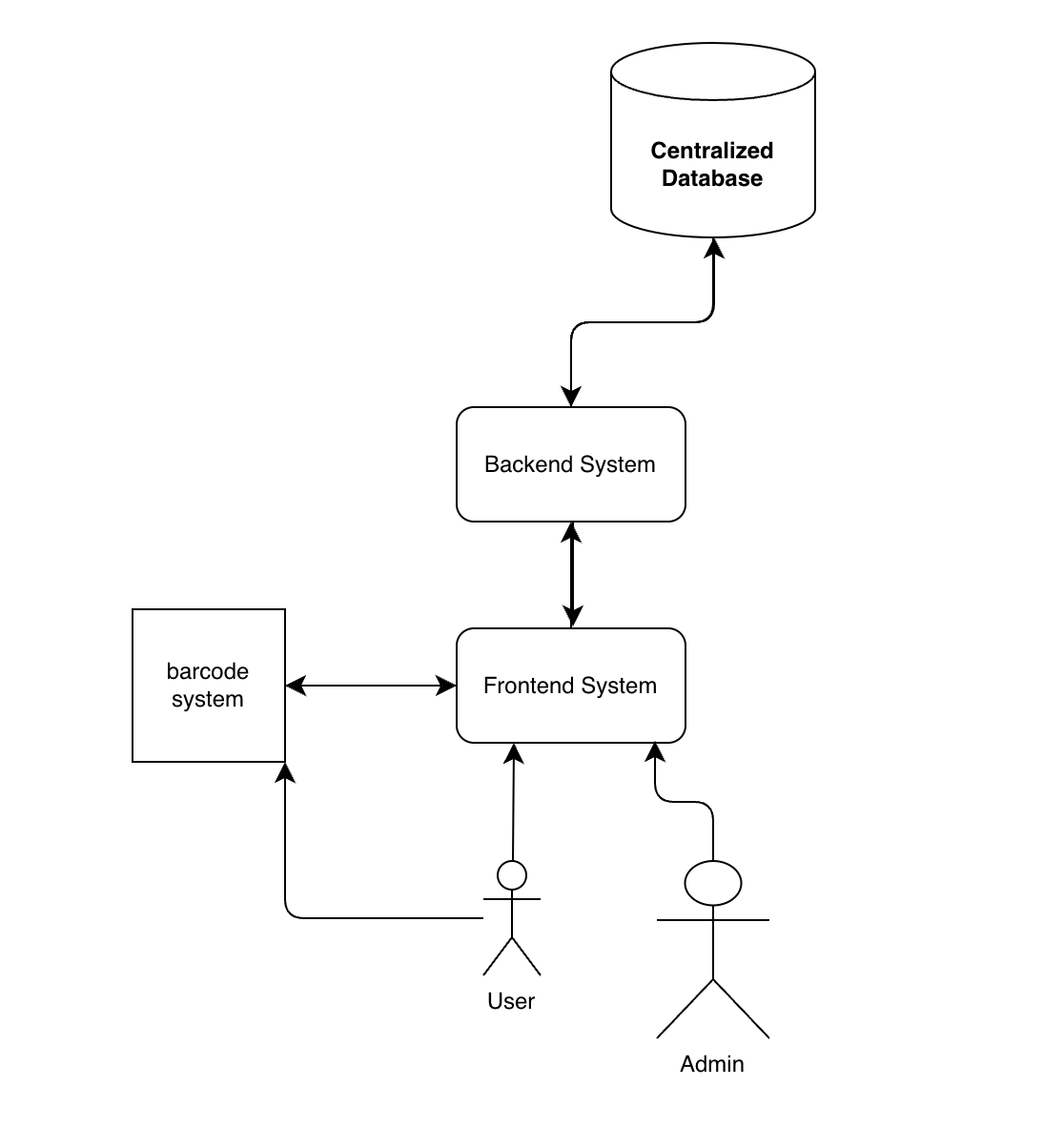
* + **Traceability and Logging:**

Logging of all system activities to ensure traceability, facilitating quick identification and resolution of any issues. Logs capture user actions, dates, and times.

* + **Warehouse Scale Consideration:**

Adaptation to the scale of a large warehouse environment, considering the specific operational needs and challenges associated with a sizable inventory.

* **Working**:



A centralized database that houses data on orders, items, and inventory levels powers the system. A user-friendly interface facilitates interaction between users and the system. The inventory levels are updated and new products are added to the system upon procurement. Orders are handled, and prompt fulfillment is guaranteed by the system.

Barcode scanning reduces errors in product identification and makes tracking more efficient. Through the generation of reports and analytics, the system offers insights into financial performance, stock turnover, and sales trends. Accurate and current financial records are ensured by integration with accounting systems.

* **Deficiencies:**
  + **Subpar User Interface (UI):**

The user interface lacks user-friendliness, creating challenges in understanding and questionable usability. Improvements are needed to enhance the overall user experience.

* + **Potential Outdated Technology:**

There is uncertainty about the system being developed using outdated technology. If confirmed, this presents a challenge, as finding expertise for such technology may be difficult. This uncertainty hampers the system's potential for updates and improvements.

* + **Monolithic Architecture Hurdles:**

The system employs a monolithic architecture, posing challenges for customization and adaptation. Making changes to the system is complex due to its integrated structure. Exploring a more modular architecture could ease the burden of customization and updates.

* + **Security Oversight:**

One or more security perspectives may have been overlooked or inadequately addressed in the system. This creates potential vulnerabilities and risks to data security. A comprehensive security audit is necessary to identify and rectify these oversight issues, ensuring robust protection against potential threats.

## Part 2: Project Maintenance Types

Let’s go through different kinds of maintenance one by one

* **Corrective Maintenance:**

Immediate attention is required due to identified security issues posing a significant threat. Conducting a thorough security audit, documenting issues, and applying updates are crucial steps to fortify our system against potential threats.

* **Adaptive Maintenance:**

Recognizing the outdated tech stack's limitations in scalability, planning for a systematic technology upgrade is essential. Ensuring smooth integration and transition to the updated system is a priority.

* **Perfective Maintenance:**

While the current UI may suffice, proactive measures involve a comprehensive analysis and enhancement of the UI design for an optimal user experience.

* **Preventive Maintenance:**

Addressing the time-consuming impact of every change in our monolithic system involves refactoring and transitioning to a modular approach. Gradual adoption of a microservices architecture enhances manageability and adaptability.

* **Adaptability Maintenance:**

Addressing the lack of mobile support involves making the design responsive or developing a dedicated mobile app. Decisions should align with project requirements while maintaining a centralized internet-based database.

* **Routine Maintenance:**

Establishing routine check-ups for system performance and regular log reviews is crucial to prevent undesirable situations that may necessitate emergency maintenance, ensuring ongoing system reliability.

* **Emergency Maintenance:**

In the event of bugs affecting order processing or other features, prioritizing system fixes, restoring normal operation, and identifying root causes are crucial. Consistent routine maintenance plays a vital role in averting such emergencies.

## Part 03

Here's how the project changes could be prioritized according to urgent, critical, major, and minor categories:

* **Urgent Changes:**
* Immediate attention is required due to identified security issues posing significant threats. Conducting a thorough security audit and applying updates to address vulnerabilities should be the top priority.
* **Critical Changes:**
* Fixing bugs affecting critical features like order processing or data integrity issues. These changes are essential for the core functionality and reliability of the system.
* **Major Changes:**
* Upgrading the technology stack to address scalability limitations and improve system performance.
* Enhancing the user interface to improve usability and user experience.
* Refactoring the monolithic architecture to a modular approach for easier customization and updates.
* **Minor Changes:**
* Adding mobile support to the system for improved accessibility.
* Implementing routine maintenance tasks such as performance optimizations and log reviews.
* Making incremental improvements to existing features based on user feedback or evolving business requirements.

## Part 04

For an inventory management system designed for a large-scale warehouse environment, several COTS (Commercial Off-The-Shelf) systems or components can be beneficial. Here are some suggestions along with the reasons for choosing them:

* Inventory Management Software:
  + Suggested Component: Odoo Inventory Management
  + Reason: Odoo offers a comprehensive inventory management solution that is highly customizable and scalable. It provides features like product tracking, barcode scanning, order processing, and integration with other business modules like sales and purchasing. Its modular architecture aligns well with the need for scalability and adaptability in a large warehouse environment.
* User Interface Framework:
  + Suggested Component: React.js or Angular
  + Reason: Both React.js and Angular are popular frontend frameworks known for their robustness and scalability. They offer tools and libraries for building interactive user interfaces with ease. Choosing one of these frameworks would address the deficiency of a subpar user interface by allowing for the development of modern, user-friendly UI components.
* Security Framework:
  + Suggested Component: Auth0
  + Reason: Auth0 provides a comprehensive identity management platform with features like authentication, authorization, and single sign-on (SSO). It offers robust security mechanisms such as multi-factor authentication and OAuth integration, addressing the security oversight in the existing system. Implementing Auth0 would enhance user authentication and access control, ensuring data security in the inventory management system.
* Database Management System:
  + Suggested Component: PostgreSQL
  + Reason: PostgreSQL is a powerful open-source relational database management system (RDBMS) known for its reliability, performance, and scalability. It can handle large volumes of data efficiently, making it suitable for a large-scale warehouse environment. Integrating PostgreSQL would improve data management and ensure the integrity of inventory records in the system.
* Reporting and Analytics Tool:
  + Suggested Component: Power BI
  + Reason: Power BI is a robust business intelligence and analytics tool that offers powerful reporting capabilities and data visualization features. It can connect to various data sources, including databases and APIs, to generate insightful reports and dashboards. Integrating Power BI would address the need for comprehensive record maintenance and analytics, providing real-time insights into inventory performance and sales trends.

By incorporating these COTS-based systems and components into the project, the inventory management system can be enhanced in terms of functionality, usability, security, and scalability, meeting the specific requirements of a large-scale warehouse environment.

**Part 5: Apply Maintenance Process**  
Apply maintenance process on your project’s suggested problems and changes. You can apply IEEE/EIA 1219 standard maintenance process or ISO/IEC 14764 standard maintenance process on your project.

**1. Problem Identification and Reporting:**

Subpar User Interface: Users find the interface difficult to navigate and understand, leading to frustration and inefficiency. Examples: confusing menu layouts, inconsistent iconography, unclear error messages.

Outdated Technology Stack: The system relies on aging technologies, limiting its scalability, security updates, and integration with newer systems. Examples: outdated programming languages, lack of support for modern operating systems.

Monolithic Architecture: Making changes to the system is complex and time-consuming due to its tightly coupled components. This inflexibility hinders customization and adaptation to evolving business needs. Examples: adding new features requires modifying core code, changes in one module impact others.

Security Vulnerabilities: Potential security weaknesses could expose sensitive data to unauthorized access or manipulation. Examples: insufficient user authentication, lack of data encryption, unpatched vulnerabilities in underlying libraries.

**2. Problem Evaluation:**

Severity:

Security vulnerabilities: Critical: Immediate action required to prevent potential data breaches.

Outdated technology stack: High: Long-term risk of system instability and inability to meet future needs.

Monolithic architecture: Medium: Impacts development agility and adaptability to changing requirements.

Subpar UI: Low-Medium: Decreases user productivity and satisfaction, but not critical for system functionality.

Urgency:

Security vulnerabilities: Urgent: Patching and security improvements must be implemented immediately.

Outdated technology stack: Important, schedule migration plan based on resource availability.

Monolithic architecture: Gradual transition to modular architecture over time.

Subpar UI: Improvements can be phased based on priority and available resources.

Resources:

Estimate personnel, time, and budget required for each solution, considering complexity and technology choices.

**3. Solution Recommendation:**

Subpar UI:

Conduct user research and gather feedback to identify specific pain points.

Redesign the interface for improved usability and visual appeal, adhering to design principles and user-centered practices.

Develop prototypes and conduct user testing to iterate and refine the design.

Outdated Technology Stack:

Research and identify a modern technology stack suitable for your project needs and existing infrastructure.

Develop a phased migration plan, outlining steps for transitioning to the new technology stack with minimal disruption.

Ensure compatibility and data migration strategies between old and new systems.

Monolithic Architecture:

Start by identifying core functionalities and potential boundaries for microservices.

Gradually refactor code modules into independent services with well-defined APIs for communication.

Implement infrastructure and orchestration tools to manage and deploy microservices efficiently.

Security Vulnerabilities:

Conduct a comprehensive security audit to identify vulnerabilities in code, configuration, and network infrastructure.

Prioritize critical vulnerabilities and implement patches or mitigating controls to address them immediately.

Review and strengthen security practices, including user authentication, data encryption, and access control policies.

**4. Solution Approval:**

Present the proposed solutions to stakeholders, including developers, users, and management.

Discuss the trade-offs, risks, and benefits of each solution in the context of project goals and resource constraints.

Obtain formal approval and commitment from stakeholders before proceeding with implementation.

**5. Implementation and Testing:**

Develop and implement the approved solutions in phases, prioritizing critical issues first.

Utilize appropriate development methodologies and tools like testing frameworks and version control systems.

Conduct thorough testing throughout the implementation process, including unit tests, integration tests, and system-level testing.

Fix any bugs or issues identified during testing before deploying the changes to production environments.

**6. Maintenance and Review:**

Monitor the system after implementation to identify any unexpected issues or performance degradation.

Review the maintenance process periodically and adapt it as needed based on lessons learned and new challenges.

Schedule regular maintenance activities like security audits, system backups, and performance optimization.

**Part 6: Selection of Quality Attributesfor Architecture Recovery**

Suggest any 4 quality attributesthat you think they should be present in the system for recovering the architecture.

**Selecting Quality Attributes for Architecture Recovery:**

To successfully recover the architecture of your Inventory Management System, prioritizing the **following four quality attributes is crucial:**

**1. Maintainability:**

A maintainable system enables ongoing modifications, bug fixes, and feature enhancements with minimal effort and disruption. Here's how we can prioritize maintainability in your architecture recovery:

Modular architecture: Break down the system into independent, loosely coupled modules with well-defined interfaces. This allows for isolated changes and updates without affecting other parts of the system.

Clean code: Use well-organized code with consistent formatting, meaningful variable names, and thorough documentation. This makes the code easier to understand, modify, and debug.

Automated testing: Implement automated unit and integration tests to catch regressions and ensure code quality. This saves time and reduces the risk of introducing new bugs during maintenance.

Configuration management: Use version control systems like Git to track code changes and revert to previous versions if needed. This simplifies rollbacks and maintains version history.

Dependency management: Clearly define and manage dependencies between modules to avoid conflicts and unexpected behavior during updates.

**2. Scalability:**

The recovered architecture should be able to handle increasing data volumes and user demands without performance degradation. Consider these principles:

Horizontal scaling: Design the system to leverage cloud-based infrastructure for horizontal scaling of individual components. This allows you to add more resources (like servers) as needed.

Microservices architecture: Utilize microservices architecture where each service can be scaled independently based on its workload and resource requirements.

Database optimization: Choose a scalable database solution that can efficiently handle large amounts of data and concurrent transactions.

Caching mechanisms: Implement caching mechanisms to reduce database load and improve response times for frequently accessed data.

**3. Interoperability:**

The system should seamlessly integrate with existing systems and platforms, fostering data exchange and collaboration. Focus on these aspects:

Open standards: Use open standards and APIs for data exchange and communication with other systems. This ensures compatibility and avoids vendor lock-in.

Data serialization formats: Choose standardized data formats like JSON or XML for data exchange between the system and external platforms.

Event-driven architecture: Consider adopting an event-driven architecture where services communicate through asynchronous events, ensuring loose coupling and flexibility.

**4. Robustness:**

The system should be resilient to errors and failures, minimizing downtime and data loss. Emphasize these points:

Fault tolerance: Implement fault tolerance mechanisms like redundancy and failover to prevent single points of failure and ensure uptime during component outages.

Error handling: Design robust error handling mechanisms to gracefully handle exceptions and prevent cascading failures.

Logging and monitoring: Implement comprehensive logging and monitoring systems to track system health, identify potential issues, and facilitate debugging.

Backup and recovery: Establish regular backups of system data and a reliable disaster recovery plan to restore operations in case of major failures.

**Functionality 1: Add Product to Inventory**

**Original Code:**

import java.util.ArrayList;

class Inventory {

private ArrayList<Product> products;

public Inventory() {

this.products = new ArrayList<>();

}

public void addProduct(Product product) {

products.add(product);

}

}

class Product {

private int id;

private String name;

private int quantity;

private double price;

// Constructor, getters, setters

}

// Usage

Product product1 = new Product(1, "Product A", 100, 10.99);

Inventory inventorySystem = new Inventory();

inventorySystem.addProduct(product1);

```

**Refactored Code:**

import java.util.ArrayList;

class Inventory {

private ArrayList<Product> products;

public Inventory() {

this.products = new ArrayList<>();

}

public void addProduct(Product product) {

products.add(product);

}

}

class InventorySlicer {

public void sliceAddProduct(Product product) {

Inventory inventorySystem = new Inventory();

inventorySystem.addProduct(product);

}

}

// Usage

Product product1 = new Product(1, "Product A", 100, 10.99);

InventorySlicer slicer = new InventorySlicer();

slicer.sliceAddProduct(product1);

**Explanation:**

**1. Original Code:** Defines an `Inventory` class with an `addProduct` method for adding a product to the inventory.

**2. Refactored Code:** Introduces a new class `InventorySlicer` with a method `sliceAddProduct`. This method creates an instance of `Inventory` and adds a product, isolating the functionality of adding a product.

**Program Slicing for `addProduct` Functionality:**

class Inventory {

public void addProduct(Product product) {

products.add(product);

}

}

// Usage

Product product1 = new Product(1, "Product A", 100, 10.99);

Inventory inventorySystem = new Inventory();

inventorySystem.addProduct(product1);

```

**Explanation:**

1. Program Slicing: Extracts only the relevant code for the `addProduct` functionality, excluding unnecessary parts of the original code.

**Functionality 2: Update Product Quantity**

**Original Code:**

import java.util.ArrayList;

class Inventory {

private ArrayList<Product> products;

public Inventory() {

this.products = new ArrayList<>();

}

public void updateQuantity(int productId, int newQuantity) {

for (Product product : products) {

if (product.getId() == productId) {

product.setQuantity(newQuantity);

break;

}

}

}

}

class Product {

private int id;

private String name;

private int quantity;

private double price;

// Constructor, getters, setters

}

// Usage

Product product1 = new Product(1, "Product A", 100, 10.99);

Inventory inventorySystem = new Inventory();

inventorySystem.addProduct(product1);

inventorySystem.updateQuantity(1, 150);

**Refactored Code:**

import java.util.ArrayList;

class Inventory {

private ArrayList<Product> products;

public Inventory() {

this.products = new ArrayList<>();

}

public void updateQuantity(int productId, int newQuantity) {

for (Product product : products) {

if (product.getId() == productId) {

product.setQuantity(newQuantity);

break;

}

}

}

}

class InventorySlicer {

public void sliceUpdateQuantity(int productId, int newQuantity) {

Inventory inventorySystem = new Inventory();

for (Product product : inventorySystem.getProducts()) {

if (product.getId() == productId) {

product.setQuantity(newQuantity);

break;

}

}

}

}

// Usage

Product product1 = new Product(1, "Product A", 100, 10.99);

Inventory inventorySystem = new Inventory();

inventorySystem.addProduct(product1);

InventorySlicer slicer = new InventorySlicer();

slicer.sliceUpdateQuantity(1, 150);

**Explanation:**

**1. Original Code:** Includes a method `updateQuantity` within the `Inventory` class to update the quantity of a product based on its ID.

**2. Refactored Code:** Introduces a new class `InventorySlicer` with a method `sliceUpdateQuantity`. This method creates an instance of `Inventory`, iterates through its products, and updates the quantity of a product based on the provided ID.

**Program Slicing for `updateQuantity` Functionality:**

class Inventory {

public void updateQuantity(int productId, int newQuantity) {

for (Product product : products) {

if (product.getId() == productId) {

product.setQuantity(newQuantity);

break;

}

}

}

}

// Usage

Product product1 = new Product(1, "Product A", 100, 10.99);

Inventory inventorySystem = new Inventory();

inventorySystem.addProduct(product1);

inventorySystem.updateQuantity(1, 150);

```

**Explanation:**

**Program Slicing:** Extracts only the relevant code for the `updateQuantity` functionality, excluding unnecessary parts of the original code.

Part 8: Advantages of Re-engineering and Summary

Software re-engineering offers several advantages for a project, contributing to its long-term success and sustainability.

**Advantages of Re-engineering:**

**1. Enhanced Maintainability:**

Re-engineering improves the overall structure of the software, making it more modular and easier to maintain.

Refactored code and program slicing help in isolating and addressing specific functionalities, reducing complexity.

**2. Improved Performance:**

Through refactoring, inefficient parts of the code can be optimized, resulting in improved performance.

Program slicing allows for targeted modifications, ensuring that changes do not adversely affect the system's performance.

**3. Adaptation to Changing Requirements:**

Software re-engineering allows for the incorporation of new features and functionalities to meet changing business requirements.

Applying the maintenance process helps in addressing evolving needs effectively.

**4. Reduced Defects and Technical Debt:**

The identification and elimination of deficiencies through maintenance activities lead to a reduction in defects.

Refactoring helps in managing technical debt by addressing code complexities and ensuring a more reliable software foundation.

**5. Cost-Efficiency:**

Re-engineering enables cost-effective solutions by eliminating unnecessary functionalities, optimizing code, and utilizing COTS-based systems/components.

Prioritizing changes based on urgency and severity allows for efficient resource allocation.

**Summary of Outcomes:**

In summary, the re-engineering activities in this project have focused on improving maintainability, performance, and adaptability to changing requirements. The project team applied maintenance processes, selected appropriate COTS components, and prioritized changes effectively. The use of refactoring and program slicing further strengthened the codebase and targeted specific functionalities for improvement.

**Overall Project Impact:**

The re-engineering efforts have positively impacted the project's overall quality, making it more resilient to future changes and scalable. The combination of IEEE/EIA 1219 and ISO/IEC 14764 standards provided a robust framework for maintenance activities, ensuring systematic and well-documented processes.

The advantages gained from software re-engineering contribute to the project's long-term success by aligning it with current best practices, improving its ability to adapt to evolving needs, and reducing technical debt. The team's efforts have resulted in a more maintainable, efficient, and reliable software system.